eProsima Dynamic Fast Buffers

User Manual
Version 0.2.0

The Middleware Experts

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 2 – 1ºG
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Trademarks

License

Technical Support

- Phone: +34 91 804 34 48
- Email: support@eProsima.com

Table of Contents

eProsima Dynamic Fast Buffers....1
1 Introduction....4
1.1 Data Describing and Serializing....4
1.2 Main Features....5
2 Building an application....7
2.1 Describing data through a Typecode....8
2.1.1 Typecode creation syntax....8
2.2 Generating a specific Bytecode given a Typecode object....15
2.2.1 Bytecode for data serialization:....15
2.2.2 Bytecode for data deserialization:....16
2.3 Data serialization and deserialization....16
2.3.1 Data serialization:....16
2.3.2 Data deserialization....17
3 eProsima Dynamic Fast Buffers API....18
3.1 Typecode creation API....18
3.2 Bytecode generation API....18
3.3 Data serialization/deserialization API....18
4 Known Issues....19
5 HelloWorld example in Visual Studio 2010....20
5.1 Setting up the environment....20
5.2 Including headers....20
5.3 Data declaration and “FastCdr” object creation....20
5.4 Typecode creation....21
5.5 Bytecode generation....21
5.6 Data serialization....22
5.7 Buffer reset....22
5.8 Data deserialization and Typecode destruction....22

1 Introduction

eProsima Dynamic Fast Buffers is a high-performance dynamic serialization library, which allows users to describe and serialize or deserialize data at run-time. Its functionality is based on the creation of a typecode which defines the data, and the generation afterwards of a bytecode to do the data serialization. There is no need for users to know anything about the internal structure of the typecode or the bytecode, neither of the serialization procedure.

This library uses *eProsima Fast CDR,* which is a library designed to serialize and deserialize data in CDR (Common Data Representation) format. CDR is a transfer syntax low-level representation for transfer between agents, which describes a mapping from data types defined in OMG IDL (Interface Definition Language) to a byte stream.

IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language-independent way, enabling communication between software components that do not share the same language.

1.1 Data Describing and Serializing

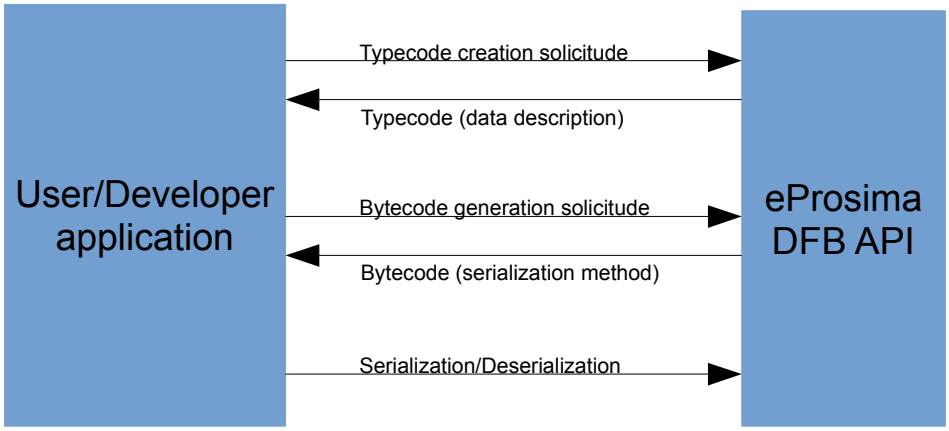
In computer science, inside the context of data storage and transmission, serialization is the process of translating data structures or objects' state into a format that can be stored (for example in a file or a memory buffer) and recovered later in the same or another computer environment.

When it comes to dynamic serialization, the native data has to be described somehow. It is in this moment when *eProsima Dynamic Fast Buffers* comes in, providing the functionality for doing it.

The data description may be done by using a typecode. This is just an entity that stores information in order to provide full knowledge of the data described through it, taking care only of the data type and not of its content or value. This avoids developers from defining their data types inside an IDL file that would have to be parsed later at run-time.

Once the data has been described, the application gets full knowledge of it and of how to manage its content. At this moment, a bytecode can be generated in order to define how this data types must be serialized, therefore, it is an internal definition of how to perform this serialization.

The image below shows the library's functionality:



1.2 Main Features

eProsima Dynamic Fast Buffers (DFB from now on) provides an easy way to describe and serialize or deserialize native data which has been defined by the user. It brings along these features:

- **Data description through a typecode:**
    - A typecode is a way to describe how is the data that a user wants to use in his application.
    - Through this typecode, *eProsima DFB* knows how is the user's native data and how to move through it.
    - Avoids the developer from describing data in a static way using an IDL file.
- **Bytecode generation:**
    - This is just a way for *DFB* of knowing how to serialize the native data defined by the user.
    - There are two kinds of bytecode entities that can be generated: one for data serialization, and another for data deserialization.
- **Data serialization/deserialization**:
    - *eProsima DBF* features a way to serialize or deserialize data using a *FastCDR* object provided by *eProsima Fast CDR* library.
    - The serialized data will be stored inside a buffer defined by the user.
    - The deserialized data will be restored in the user's native data type previously defined.
- **Support for different packaging strategies for structures:**
    - *eProsima Dynamic Fast Buffers* supports different packaging strategies which can be selected by the user at compile time. This means that if it comes the time when the user wants to pack the contents of his structures by adding no padding, this library will still work.
        - *eProsima DFB* has been tested using all available boundaries for the native data types.
        - This feature will only work in Windows, for in Linux this can only be achieved using *-fpack-struct* option that gcc provides, and this is only compatible with libraries compiled using that same flag.
        - The use of *pragma* directives (designed to specify different alignment for structures) is not supported, because of the dynamic behavior of this library.

- **Multi-platform:**
    - *eProsima Dynamic Fast Buffers* has been designed and tested for different platforms. It supports Windows and Linux (Fedora and CentOS distributions) Operating Systems, in both 32-bit and 64-bit architectures.

2 Building an application

eProsima Dynamic Fast Buffers allows a developer to easily describe its own native data types using a typecode. It provides functions to serialize these data into a previously created buffer and deserialize them back.

How the library defines or works with the users' data types must be transparent, so there is no need for them to know any of that information. From the developer's point of view, a *Typecode* object that represents the data can be created in his application, and this object could be used afterwards for creating a *Bytecode* object.

This object will be used later to know how to serialize the user's data. In the same way, how *eProsima DFB* uses this *Bytecode* object should be transparent for the developer, for only the data description concerns to him.

eProsima DFB offers this transparency and facilitates the development. The general steps to build an application which uses *DFB* library are these:

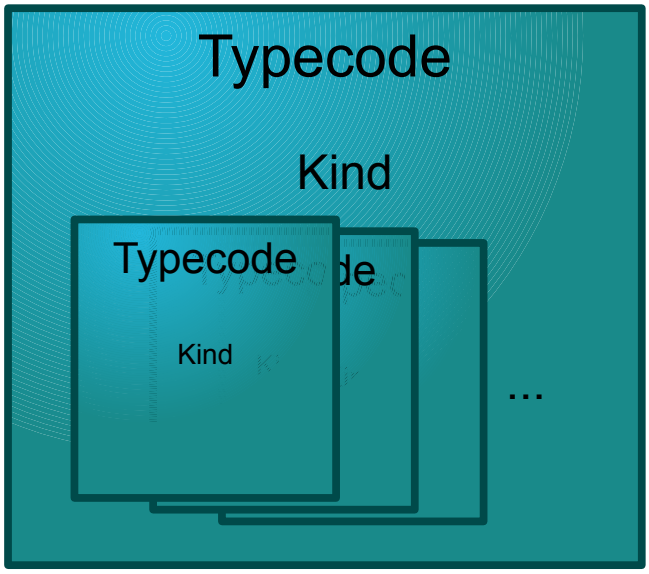
- Create a *Typecode* object that represents the data.
- Generation of a *Bytecode* object so that the library knows how to serialize or deserialize the mentioned data.
- Serialize data into a *FastBuffer* object using the provided *eProsima DFB* API for that task.
- Deserialize the data previously serialized into the user's native structures (simple or complex).

This section describes the basic concepts of these four steps that a developer has to follow to use *eProsima DFB*.

2.1 Describing data through a Typecode

A *Typecode* object is used by *eProsima DFB* to describe how the user's native data in his application is. That is why this object must be created according to the data that will be serialized later. If this definition is made wrong, *eProsima DBF* does nor guarantee a successful execution.

eProsima DFB's typecode is defined by using a *kind* attribute (which specifies the data type) and it may contain other *Typecode* objects inside it. The reason for that representation is to provide the developers with a way to define complex data types such as structures, inserting other simple or complex data descriptions inside them.



2.1.1 Typecode creation syntax

2.1.1.1 Simple types supported

eProsima DFB supports a complete variety of simple types that can be described by the developer via the provided API functions. The following table shows the supported types and the functions that must be called to create them.

TABLE 1: SPECIFYING SIMPLE TYPES IN IDL FOR C++

User Type	C++ Sample Data Type	Function for creation
char	`char charSample;`	`createCharacter()`
short	`int16_t shortSample;`	`createShort()`
unsigned short	`uint16_t shortSample;`	`createShort()`
int	`int32_t intSample;`	`createInteger()`
unsigned int	`uint32_t intSample;`	`createInteger()`
long	`int64_t longSample;`	`createLong()`
unsigned long	`uint64_t longSample;`	`createLong()`
float	`float floatSample;`	`createFloat()`
double	`double doubleSample;`	`createDouble()`
string	`std::string stringSample;`	`createString()`
bool	`bool boolSample;`	`createBoolean()`

2.1.1.2 Complex types supported

Complex types can be created by the developer containing simple types or other complex types (if supported). These complex types can be used as containers for other simple types or just to define complex data structures such as arrays. The following table shows the supported type descriptions (typecodes), how they are defined in C++ programming language, and which is the function for creating them.

TABLE 2: SPECIFYING COMPLEX TYPES IN IDL FOR C++

User Type	C++ Sample Data Type	Function for creation
struct (see note below)	`struct structSample { ... };`	`createStruct(type1, type2, ..., typeN, NULL)`
array	`std::array<type, size> arraySample1(); type arraySample2[][]...[];`	`createArray(type, nDims, dim1, dim2, ..., dimN, 0)`
bounded sequence	`std::vector<type> sequenceSample(size);`	`createSequence(type, size)`

Note: Structures may contain any kind of data type, but the other complex types cannot contain complex data types.

2.1.1.3 Simple data types description

The functions used for doing simple data description are defined in this section. This will be done through the creation of a typecode associated to the mentioned data. These functions are used always the same way, and they have been designed to provide the users with a simple API for data definition.

The functions of this API are defined in the class *DynamicFastBuffers::TypecodeAPI*.

Character data definition:

To describe a character data type, the function *createCharacter* must be used. By calling it, the users will be able to obtain a *Typecode* object which describes this kind of data.

This function is executed as it is shown below:

```
DynamicFastBuffers::Typecode *characterTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createCharacter();
```

By using this function, a new object that defines a character data type will be allocated in the previously declared pointer.

Short data definition:

In order to describe a short data type for being serialized, the function named *createShort,* included in the TypecodeAPI class, must be used.

This function is shown in the following code sample:

```
DynamicFastBuffers::Typecode *shortTypecode;
shortTypecode = DynamicFastBuffers::TypecodeAPI::createShort();
```

Inside the object pointed by *shortTypecode*, there will be an instance of *Typecode* whose kind represents a short data type.

Integer data definition:

To define a typecode which represents an integer value defined by the user, the function *createInteger* must be used.

```
DynamicFastBuffers::Typecode *integerTypecode;
integerTypecode = DynamicFastBuffers::TypecodeAPI::createInteger();
```

Long data definition:

In order to describe a long data type, the function *createLong* has to be used. An example of how to use the mentioned function is shown below:

```
DynamicFastBuffers::Typecode *longTypecode;
longTypecode = DynamicFastBuffers::TypecodeAPI::createLong();
```

Float data definition:

To describe a simple precision floating point number, a function named *createFloat* must be used. This function is also defined in the *TypecodeAPI* class, and it can be accessed through *DynamicFastBuffers* namespace.

The next example shows how to create a *Typecode* object associated to a float data type:

```
DynamicFastBuffers::Typecode *floatTypecode;
floatTypecode = DynamicFastBuffers::TypecodeAPI::createFloat();
```

Double data definition:

The function used for describing a double precision floating point number is named *createDouble*, and as the other functions it can be accessed via the *TypecodeAPI* class.

```
DynamicFastBuffers::Typecode *doubleTypecode;
doubleTypecode = DynamicFastBuffers::TypecodeAPI::createDouble();
```

String data definition:

By using *eProsima DFB*, the user can also describe objects defined using std::string. The function for describing this kind of objects is named *createString*.

In this example, the way to describe an std::string object by using eProsima DFB Typecode API is shown.

```
DynamicFastBuffers::Typecode *stringTypecode;
stringTypecode = DynamicFastBuffers::TypecodeAPI::createString();
```

Boolean data definition:

If any user wants to describe a boolean data type for serializing or deserializing it, the *TypecodeAPI* class provides a function named *createBoolean* to do it. This function is used as it is shown below:

```
DynamicFastBuffers::Typecode *boolTypecode;
boolTypecode = DynamicFastBuffers::TypecodeAPI::createBoolean();
```

2.1.1.4 Complex data types definition

On the other hand, if the user wants to describe complex data types, another set of functions has to be used. These other functions have parameters that the mentioned user must know in order to use them, which is a clear difference against the ones described in the previous section.

There are three kinds of data types that can be defined though a typecode by using *eProsima DFB*. These kinds are:

- Structures
- Arrays
- Sequences

Structure data definition:

When it comes to describing struct data types, the user must bear in mind that this kind of types are composed by more simple or complex data types. This means that there will be inner *Typecode* objects inside the external one used to describe the structure.

In this case, the function used to create the typecode for the structure (whose name is *createStruct)* is a little bit different from the previous ones. Nevertheless, the way to use it is similar.

There are two main paths that may be chosen to describe a structure typecode. The first one is to create the object and then, using a function named *addMembers,* specify one or more *Typecode* objects that will be inserted inside it. The second one is based on adding the inner typecode definitions as parameters in the *createStruct* function.

Now an example of each approach is shown:

- Creation using *createStruct* and then *addMembers*:

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct();

DynamicFastBuffers::TypecodeAPI::addMembers(
    structTypecode,
    DynamicFastBuffers::TypecodeAPI::createInteger(),
    NULL
);
```

In the previous image, a structure is described containing an integer inside it. First, an object instance of *Typecode* is created, and then the function *addMembers* is used to add an integer data type inside it. If the user wants to add more data definitions to the structure, they can be added later by calling the same function (bearing in mind this means adding them immediately after the ones that are already added).

In case a wrong call to this function is made by not specifying the destination *Typecode* object of the structure, a *WrongParamException* object will be thrown. On the other hand, if the user does not provide any *Typecode* objects to insert, an exception object instance of *NotEnoughParamsException* will be

thrown. Finally, if the destination *Typecode* object is not a structure type description, the library will rise a *WrongTypeException*.

- Creation using only *createStruct*:

  The other way of describing a structure is by adding the inner data types as parameters when creating the object. An example of how to do this can be seen in the next image:

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
    DynamicFastBuffers::TypecodeAPI::createInteger(),
    DynamicFastBuffers::TypecodeAPI::createString(),
    DynamicFastBuffers::TypecodeAPI::createStruct(
        DynamicFastBuffers::TypecodeAPI::createShort(),
        DynamicFastBuffers::TypecodeAPI::createDouble(),
        NULL
    ),
    NULL
);
```

  Developers can add other *Typecode* objects (which may be created earlier and then inserted in any order) when creating the typecode for describing the structure data type. In this example, a structure containing an integer, a string and another structure has been described.

Either the user chooses to use *addMembers* or the default function *createStruct* with parameters to insert more data types into a structure definition, it is important to know that the order of the insertions is determinant for the *Typecode* creation to be successful. This means that the order must be the same as the order of the native data in the user's application, otherwise serialization may fail.

Note that every call to createStuct or addMembers must have as last parameter a NULL value, for there is no way of knowing how many objects the user wants to insert.

Array data definition:

Other complex data types that can be described are the arrays. An array is described by the kind of data that it holds inside, and the length of its dimensions.

For example, an integer matrix with two rows and three columns will be described using a *Typecode* object and specifying that it has two dimensions, having the first one a length of two and the second one a length of three.

This concrete *Typecode* object can be created using the function named *createArray*, defined in the *TypecodeAPI* class. This function receives as parameters a *Typecode object* indicating the kind of data that will be stored inside it, followed by an integer which specifies the number of dimensions. After that, the length of each dimension has to be provided.

In the next image, a creation of a *Typecode* object that represents an array can be seen as an example. The first parameter defines the type of data that the array will hold, in this case long (64-bit integer) values. Afterwards, the first integer (two in this case) specifies how many dimensions will be provided, being the next two integers the

respective lengths of each dimension. Finally, a zero value must be inserted in order to know that no more dimensions are going to be specified.

```
DynamicFastBuffers::Typecode *arrayTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createArray(
    DynamicFastBuffers::TypecodeAPI::createLong(),
    2, 2, 3,
    0
);
```

In this function, if the number of dimensions (first integer) is lower than one, an object instance of *NotEnoughParamsException* will be thrown. On the other hand, if any of the dimension's length is lower than one (same case described before), a *WrongParamException* exception object will be thrown. Finally, an object instance of *NotEnoughParamsException* will be thrown if the number of dimensions defined is not equal to the number of dimensions really inserted by the user.

Bear in mind that only simple types are allowed inside arrays, but not structures, sequences or other arrays.

Sequence data definition:

The last complex data that *eProsima DFB* allows to describe are sequences. This kind of type is represented in C++ as a vector of objects. These objects have to be simple data types (except for std:string, which is not supported in the current version), and they cannot be under any circumstance complex types.

A sequence is defined by using a typecode object for describing the kind of data that will be stored inside, and an integer greater than zero which specifies the maximum length of the vector that will hold the data. The function used to describe this kind of data type is named *createSequence*, and it is also defined in *TypecodeAPI* class.

```
DynamicFastBuffers::Typecode *sequenceTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createSequence(
    DynamicFastBuffers::TypecodeAPI::createShort(),
    20
);
```

In this example, a sequence of twenty (at the most) short values is described. The first parameter of the *createSequence* function is used to determine what kind of data type is stored in the sequence, and the second one is the maximum number of data that can be inserted in it.

2.1.1.5 Data description destruction

Once the user has created the typecode for the description of the native types used in his application, the *Typecode* objects must be destroyed in order to not waste memory.

For doing this, a function named *deleteTypecode* is provided, which eliminates all data reserved inside the *Typecode* object.

In case of complex data types (structures, arrays and sequences), a single call to this function giving as parameter the upper typecode will be enough to erase all reserved space.

```
DynamicFastBuffers::Typecode *sequenceTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createSequence(
    DynamicFastBuffers::TypecodeAPI::createShort(),
    20
);

DynamicFastBuffers::TypecodeAPI::deleteTypecode(sequenceTypecode);
```

2.1.1.6 Calculating serialized data size

For calculating the size of the data before the serialization process, so that the buffer could be initialized properly, a function named *checkSerializedDataSize* can be called.

This function receives a pointer to a *Typecode* object, and from this object the needed buffer size will be calculated. If more than one *Typecode* object has been created, the size of all of them must be added and stored in a variable, using it afterwards for the buffer creation.

```
DynamicFastBuffers::Typecode *sequenceTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createSequence(
    DynamicFastBuffers::TypecodeAPI::createShort(),
    20
);

int size = DynamicFastBuffers::TypecodeAPI::checkSerializedDataSize(typecode);
```

In case of std::string objects, which is a variable length data type, a fixed length of 255 Bytes will be used. If the size of the string object is greater than this value, *eProsima DFB* does not guarantee that there will be enough reserved memory.

2.1.1.7 Example

Later in this document there will be added different examples for bytecode generation and data serialization and deserialization. Due to this, a *Typecode* example object is now defined, and this is the one that will be used from now on.

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
    DynamicFastBuffers::TypecodeAPI::createInteger(),
    DynamicFastBuffers::TypecodeAPI::createString(),
    DynamicFastBuffers::TypecodeAPI::createStruct(
        DynamicFastBuffers::TypecodeAPI::createShort(),
        DynamicFastBuffers::TypecodeAPI::createDouble(),
        NULL
    ),
    NULL
);
```

As it can be seen in the previous image, the typecode describes a structure with three kinds of data types inside it, an integer, an std::string object and another structure. This second structure has a short and a double data types inside.

2.1.1.8 Limitations

The limitations for the typecode creation that must be deeply considered are the following:

- While structure data types can have any other types inside, arrays and sequences cannot have complex data types, neither string or boolean types (not supported by now).
- Union or enum data types cannot be described using this version of *eProsima Dynamic Fast Buffers*.

2.2 Generating a specific Bytecode given a Typecode object

Once the typecode for describing a concrete data type is created, the generation of a bytecode associated to it is necessary to serialize data.

There are two kinds of bytecode entities that could be generated, one for doing data serialization and another for data deserialization. This is so because the functions to perform the operations are not the same, therefore a specific bytecode must be created for each one.

The API class that holds the functionality for generating *Bytecode* objects is *DynamicFastBuffers::BytecodeAPI*.

2.2.1 Bytecode for data serialization:

As it has been mentioned before, a specific bytecode for data serialization must be created, different than the bytecode for data deserialization.

eProsima DFB provides a simple API for performing this task, by executing a mere function in which users must tell the operation they want to execute by using a parameter. Once the *Typecode* object is already available, the generation of a *Bytecode* object must be done in the following way:

```
DynamicFastBuffers::Bytecode *bytecode;
bytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
    structTypecode,
    DynamicFastBuffers::flag::SERIALIZE
);
```

For the example shown in the last image, the typecode which describes the data defined in the previous chapter has been used. The code above shows a call to a function named *generateBytecode*, whose parameters are:

- structTypecode: The *Typecode* object already created.
- DynamicFastBuffers::flag::SERIALIZE: Constant value used to specify that the generated bytecode is for doing data serialization.

By executing this function, the user receives an object instance of *Bytecode* class which contains an internal structure. This structure is a list of pointers to the functions of *eProsima Fast Buffers* API that must be executed for serializing the native data.

If a NULL value is inserted as first parameter, an exception will occur. This exception is an object defined in this library, belonging to *WrongParamException* class.

2.2.2 Bytecode for data deserialization:

The same way a bytecode is generated for serializing data must be done for deserializing it. The function of the *BytecodeAPI* class that must be executed is the same one that has been executed in the last example, but specifying a different flag.

An example of how to do this will be shown now:

```
bytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
    structTypecode,
    DynamicFastBuffers::flag::DESERIALIZE
);
```

In the last example, the same *Bytecode* object has been used. This can also be done, but bearing in mind that its internal data will be overwritten, so it will not be valid for doing data serialization.

The main difference between this two function calls is only the flag specified. In this case the parameters are the following:

- structTypecode: The *Typecode* object previously created.
- DynamicFastBuffers::flag::DESERIALIZE: Constant value used to specify that the generated bytecode is for doing deserialization.

By executing this function, the user receives an object instance of the *Bytecode* class which contains an internal structure. This structure is a list of pointers to the functions of *eProsima Fast CDR* API that must be executed for deserializing this kind of data.

If a NULL value is inserted as first parameter, a *WrongParamException* will be thrown.

2.3 Data serialization and deserialization

eProsima DFB provides an API that can be easily used for data serialization and deserialization. For this matter, there are two functions defined inside the class named *DynamicFastBuffers::SerializerAPI*.

In order to do the mapping of the native data into a buffer (where each serialized datum will be stored), this types must be previously defined by coding them. For example, if any user defines an integer value (int32_t), it can be then serialized by creating a typecode description for this data, generating a bytecode then and using it for doing the mentioned serialization. The same thing happens with other data types, such as floating point numbers (float and double), structures, sequences, etc.

In the next examples, a *FastCDR* object will be used to store the serialized data. This object must be created using the following functions:

```
char buffer[500];
eProsima::FastBuffer cdrBuffer(buffer, 500);
eProsima::FastCdr cdr(cdrBuffer);
```

2.3.1 Data serialization:

When it comes to data serialization, a function named *serialize* defined in the class *DynamicFastBuffers::SerializerAPI* must be executed. This function receives as parameters a void pointer to the native data, a *Bytecode* object previously generated

using the *BytecodeAPI* member functions, and a *FastCdr* object created using *eProsima Fast CDR* library.

An example of how to perform this action is shown in the next piece of code:

```
DynamicFastBuffers::SerializerAPI::serialize(
    (void*) &data,
    bytecode,
    &cdr
);
```

As it can be seen in the image above, the *serialize* function parameters are:

- (void*) &data: Void pointer to a variable named *data*, which is the native data defined by the user.
- bytecode: A *Bytecode* object previously generated using *generateBytecode* function from the class *BytecodeAPI*.
- &cdr: *FastCdr* object created using *eProsima Fast CDR* library.

Once this operation has been performed, all the user's data must be serialized inside the buffer existent in the *FastCdr* object. To recover this data, a process of deserialization must be done, which is explained in next section.

2.3.2 Data deserialization

Concerning data deserialization, another function provided by *eProsima DFB* must be executed. This function is in the class *DynamicFastBuffers::SerializerAPI,* and its name is *deserialize*.

The next image shows an example of how to deserialize the data:

```
DynamicFastBuffers::SerializerAPI::deserialize(
    (void*) &data,
    bytecode,
    &cdr
);
```

As it can be seen in the image, the function's parameters are the same than in the *serialize* function, but the procedure is different. In this case, the data already serialized is located inside a *FastCdr* object, and the deserialization will be done into the object pointed by the *data* variable. It does not matter which is the data type of the pointed variable, as long as the typecode specified at the beginning of the execution is defined according to this data.

3 eProsima Dynamic Fast Buffers API

The API for accessing *eProsima DFB* is defined within three main classes. These classes names are *DynamicFastBuffers::TypecodeAPI*, *DynamicFastBuffers::BytecodeAPI* and *DynamicFastBuffers::SerializerAPI*.

The functions that conform the library's API are listed in next subsections. The behavior for each function will not be described, for that has already been done in previous chapters of this document.

3.1 Typecode creation API

Public functions located in *DynamicFastBuffers::TypecodeAPI*:

Function	Description
static Typecode* createCharacter()	Creates a Typecode object for a int8_t (char) native data type
static Typecode* createShort()	Creates a Typecode object for a int16_t (short) native data type
static Typecode* createInteger()	Creates a Typecode object for a int32_t native data type
static Typecode* createLong()	Creates a Typecode object for a int64_t native data type
static Typecode* createFloat()	Creates a Typecode object for a float native data type
static Typecode* createDouble()	Creates a Typecode object for a double native data type
static Typecode* createString()	Creates a Typecode object for a std::string native data type
static Typecode* createBoolean()	Creates a Typecode object for a bool native data type
static Typecode* createStruct(Typecode* init, …)	Creates a Typecode object for a structure based on the description of the inner data
static Typecode* createArray(Typecode *type, int nDims, int dim1, …)	Creates a Typecode object for an array of based on the description of its inner native data type, the dimensions and the size of each dimension.
static Typecode* createSequence(Typecode *type, int maxLength)	Creates a Typecode object for a sequence based on the description of its inner native data type and its length
static void addMembers(Typecode *dest, …)	Adds a Typecode object into a predefined structure description
static void deleteTypecode(Typecode *tc)	Deletes a Typecode object
static int checkSerializedDataSize(Typecode *tc)	Checks the size needed for serializing the data described by the Typecode object received as parameter

3.2 Bytecode generation API

Public functions located in *DynamicFastBuffers::BytecodeAPI*:

Function	Description
static Bytecode* generateBytecode(Typecode *typecode, flag flag)	Generates a Bytecode object based on a Typecode object received as parameter

3.3 Data serialization/deserialization API

Public functions located in *DynamicFastBuffers::SerializerAPI*:

Function	Description
static void serialize(void *data, Bytecode *bytecode, eProsima::FastCdr *cdr)	Serializes the data using a Bytecode object into a CDR buffer
static void deserialize(void *data, Bytecode *bytecode, eProsima::FastCdr *cdr)	Deserializes the data using a Bytecode object from a CDR buffer

4 Known Issues

The only issue that has been found while designing this library is the following:

- *eProsima Dynamic Fast Buffers* does not guarantee a correct serialization or deserialization of any data type if there is no match between the typecode created for describing the mentioned data and the native data itself.

5 HelloWorld example in Visual Studio 2010

In this section an example will be explained step by step of how to create a new project that uses *eProsima DFB* library on Visual Studio 2010. A complex structure data type will be created and represented through a typecode. This data will be serialized and deserialized in order to confirm the correct execution of the functions.

The example will be made and compiled for a 64-bit Windows 7 OS. And for that reason a 64-bit architecture compilation of the library will be needed.

5.1 Setting up the environment

The first thing that must done is to create a new project named *HelloWorldDFB* and to make sure that all references to the external libraries (in this case *eProsima Fast CDR* and *eProsima Dynamic Fast Buffers*) are correctly set.

The linked library files will be in a folder which has to be included as an additional directory. The library version will be automatically linked. Being this done, all classes and headers from this projects can be accessed.

5.2 Including headers

In the file that contains the main function, which will be the entry point to the application, some headers must be included.

```
#include “FastCdr.h”
#include “TypecodeAPI.h”
#include “BytecodeAPI.h”
#include “SerializerAPI.h”

int main()
{
    …
}
```

5.3 Data declaration and “FastCdr” object creation

Open the Visual Studio 2010 solution *HelloWorldDFB.sln*. In the file where the function named *main* is located, two tasks have to be done. The first one is to describe the native data that will be serialized, in this case a structure (outside any function, in the global scope):

```
struct HelloWorldStruct{
    short att1;
    int32_t att2;
    string att3;
};
```

This structure is formed by three inner variables, the first one is a short, the second one is int32_t and the third one is an std::string object.

The second task is to declare and instantiate, inside the *main* function, a *FastCdr* object with a *FastBuffer* inside it.

```
char buffer[500];
eProsima::FastBuffer cdrBuffer(buffer, 500);
eProsima::FastCdr cdr(cdrBuffer);
```

By coding this three lines, a buffer with five hundred bytes has been declared and inserted in the *FastCdr* object. It will be in this buffer where all the data is going to be serialized and from which all data will also be deserialized.

At this point, two objects have to be created, one for reading the data for doing the serialization procedure, and another in which the data will be recovered.

```
HelloWorldStruct inputStruct, outputStruct;
inputStruct.att1 = 10;
inputStruct.att2 = 2;
inputStruct.att3 = “Hello World!”;
```

As it can be seen in the previous image, two structures have been declared. The one named *inputStruct* is where all data will be initialized and read from. The other one, named *outputStruct* is where data will be deserialized in the end.

5.4 Typecode creation

Once the data is defined, it must be described by creating an object instance of *Typecode* class. The following code shows how to do it:

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
    DynamicFastBuffers::TypecodeAPI::createShort(),
    DynamicFastBuffers::TypecodeAPI::createInteger(),
    DynamicFastBuffers::TypecodeAPI::createString(),
    NULL
);
```

Note that the internal structure of the typecode is created with exactly the same data types and exactly in the same order than the native ones. This must be done to ensure the serialization process finishes successfully.

5.5 Bytecode generation

Now that the typecode that represents the native data defined by the user has been created, it is time for the generation of a bytecode associated to it. In this case, two *Bytecode* objects will be generated, one for data serialization and another for data deserialization. Bearing in mind that the same object can be used for both operations, but it has to be redefined for that purpose if that is the case.

```
DynamicFastBuffers::Bytecode *serializationBytecode;
DynamicFastBuffers::Bytecode *deserializationBytecode;

serializationBytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
    structTypecode,
    DynamicFastBuffers::flag::SERIALIZE
);
deserializationBytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
    structTypecode,
    DynamicFastBuffers::flag::DESERIALIZE
);
```

As it can be seen in the image above, two objects have been created. The one named *serializationBytecode* contains a few pointers to the functions used for data serialization, and the other has the pointers to the functions for deserializing the data.

5.6 Data serialization

For data serialization, a function named *serialize* which is defined within the class *DynamicFastBuffers::SerializerAPI* will be used. The code for serializing data is shown below:

```
DynamicFastBuffers::SerializerAPI::serialize(
    (void*) &inputStruct,
    serializationBytecode,
    &cdr
);
```

As it has been mentioned in previous sections, note that the data defined by the user is passed as a parameter by casting it into a void pointer, while its memory structure is described in the bytecode.

5.7 Buffer reset

If the *FastCdr* object used for deserializing data is the same that has been used for serializing it, the inner buffer of this object must be reset by executing the following function:

```
cdr.reset()
```

5.8 Data deserialization and Typecode destruction

Finally, in order to do the data deserialization into a previously declared variable, the *deserialize* function has to be invoked. The code for this call is shown next:

```
DynamicFastBuffers::SerializerAPI::deserialize(
    (void*) &outputStruct,
    serializationBytecode,
    &cdr
);
DynamicFastBuffers::TypecodeAPI::deleteTypecode(structTypecode);
```

In the image above, after the execution of the *deserialize* function, all data that has been serialized before will be recovered into the object pointed by the *outputStruct* void pointer.

To not waste memory, a final call to *deleteTypecode* function defined in *TypecodeAPI* class is done.